# AMATH 482/582: HOMEWORK 4

KATIE HIPPE

*AMATH Department, University of Washington, Seattle, WA*
*kahippe@uw.edu*

Abstract. The FashionMNIST data set contains images of various clothing items with associated labels. We may sort these images according to various classification algorithms. We have trained Fully Connected Deep Neural Networks (FCNs/DNNs) and evaluated different configurations to find the optimal method for categorizing this data set, then generalized to our previous handwritten digits data set.

## 1. Introduction and Overview

A sense of fashion is as unique as handwriting, where each person's unique style can reveal much about their character. Despite the wide range of fashions one can dress in, they are comprised of similar components, of which the FashionMNIST contains a wide database (Figure 1). These items are sorted into overall categories (trouser, pullover, sandals, bag, etc) that are assigned a unique number (0-9). These images are very complex, so we have trained a multitude of high-level Fully Connected Deep Neural Networks (FCN/DNNs) with various parameters to evaluate which hyperparameters and specific methods yielded the best results.

We first created an adaptable FCN to have adjustable parameters (number of and neurons in the hidden layers) and evaluated different combinations to get at least 85% accuracy. Then, we evaluated different configurations of inner workings, such as different optimizations, regularization, initializations, and normalizations. Finally, we further improved our model to get at least 90% accuracy for both the FashionMNIST data set as well as at least 98% accuracy for our previous MNIST handwriting data set. We compared this wish our classification methods from MNIST as well.


Figure 1. Examples of figures within FashionMNIST data set

*Date*: March 20, 2025.

## 2. Theoretical Background

We first obtained 60,000 samples of various articles of clothing, all displayed in $28 \times 28$ matrices of gray-scale pixels. We also got 10,000 additional samples with which to test our trained FCNs. Thus these samples are dimension 784, and we wish to reduce them one of (0-9). Given the complexity of the dataset, we must use more powerful nonlinear models to conduct our classifications.

Thus we turn to FCNs, a Neural Network. Neural Networks connect our initial high dimensional data to a much lower dimension set of classifications, with one or more hidden layers of different dimensionality to facilitate classification. The input layer has our original dimension (here 784) whereas the output layer will have a lowered (here 10) to represent the potential classifications.

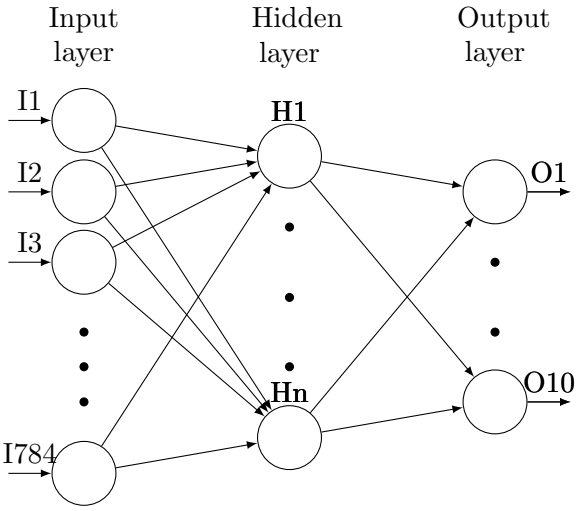



Figure 2. Example Neural Network structure

In our Neural Net, we connect the neurons (each circle in Figure 2, which acts as our computational unit) through various hidden layers. The unit is a weighted sum of its inputs, and thus the connections between neurons are a simple linear function. This sum goes through a nonlinear function to get to the next layer, which often has a different dimensions. In this case, our activation is ReLu 1, which zeros out all negative input.

$$f(x) = \max(x, 0) \tag{1}$$

As we step through the different epochs (each time looking at our data), the learning algorithm updates the weights and biases of each node until we arrive at an optimized training accuracy. At each epoch, we calculate the accuracy on our validation set, as well as compute the training loss 2, which is the error when the model makes predictions on the training data. This loss informs back-propagation to fine-tune the weights. As the model improves its classifications, this training loss should drop.

$$L(\hat{y}, y) \propto ylog(\hat{y}) + (1 - y)log(1 - \hat{y}) \tag{2}$$

FCNs have plenty of other hyperparameters with which we can control the performance of our model. For example, the number of hidden layers and the dimensions between them, as adding more or fewer hidden layers can adjust how well the model performs and how fast it runs. We can also run the model over an adjustable number of epochs (how many times we rerun our entire testing set), and batch our training data into an adjustable number of smaller batches, which will aid computational speed as we may update parameters based on a smaller subset of our data.

The learning rate, which controls how much the model changes in response to the estimated error (training loss), may also be changed and optimized manually (in this paper, between $10^{-3}$ and $10^{-1}$)

Still other hyperparameters include the type of optimizer we use to figure out the weighted sum. As we train the data, the optimizer works in tandem with the loss function to find the best set of weights to minimize the error. For example, we may use Stochastic Gradient Descent (SGD), which descends in the direction of the gradient for a subset of our data; Adam, which uses an adaptive learning rate and momentum; or RMSProp, which uses different weighted learning rates when calculating each parameter.

Related to the optimizer is the initialization of the weights. Choosing a good initialization distribution can be paramount in training convergence and speed of model training. For example, we may use Random Normal, which samples from a Normal distribution with mean 0 and standard deviation 0.1; Xavier Normal, which uses a normal distribution scaled by the number of input neurons; or Kaiming (He) Uniform, which initializes uniformly from a bounded distribution.

We may also include a regularization, which acts over the training data to prevent models from overfitting. Thus our model would work better on unseen data, as it would be protected from influence from potential outliers. Droupout regularization is an example of one such technique, in which we drop data points from the data set while training at a random probability. Thus our model will have to learn a more generalizable classification, as it'll be looking at a different subset of the neurons with each iteration.

Finally, normalization of the data at large and batch normalization both help speed and accuracy. Batch normalization acts over the activations of a layer during the training, which can stabilize any potential internal covariate shift. It can also act as a regularizer.

## 3. Algorithm Implementation and Development

To develop and run these functions and algorithms, we used Python 3.12.8 in Visual Studios Code. For general numerical methods, we used NumPy [1]. For building and running our FCNs, including managing our hyperparameters and inner functions, we used torch [3]. For plotting, we used Matplotlib [2] and Seaborn [4].

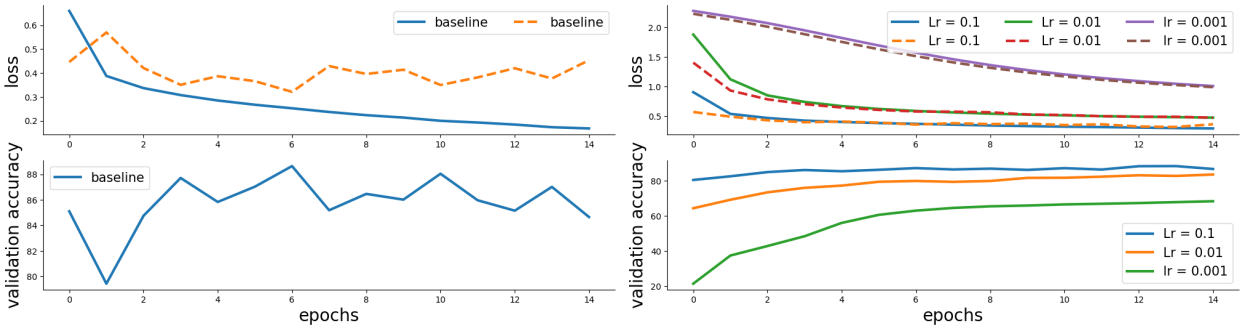
First, we developed the framework that would allow us to customize our FCNs. This involved making all our parameters customizable (hyperparameters, types of optimizers, etc). Then, for each alternate configuration, we reran the model, keeping track of the training loss and accuracy and validation accuracy to evaluate each model configuration. We graphed these based on epoch and took note of the validation accuracy as well.

## 4. Computational Results

Our first major computational result is the baseline configuration for our model. This one uses two hidden layers, the first with dimension 500 and the second with dimension 50. This model uses a learning rate (lr) of 0.1, and runs across 15 epochs. We have used Cross Entropy loss to perform the classification and the SGD optimizer to optimize our steps. This baseline is our first result in Table 1 and yields an 87.64% testing accuracy with standard deviation 0.0005. The progression of this model through its epochs is displayed in Figure 3a. For this and all following figures, the training loss is logged in a solid line while the validation loss is a dashed line.

Next, we began to test different optimizers in conjunction with different learning rates. We tested using SGD under $\text{lr} = \{0.1, 0.01, 0.001\}$ and saw how the training loss decreased and the validation accuracy increased over time, then took note of the testing accuracy. Results are displayed in Figure 3b. A higher learning rate corresponds to the weights being updated more with each epoch, which could either lead to a faster convergence or instability. In this case, we see that the highest learning rate of $\text{lr} = 0.1$ yielded the former scenario, as the model approaches a high accuracy very

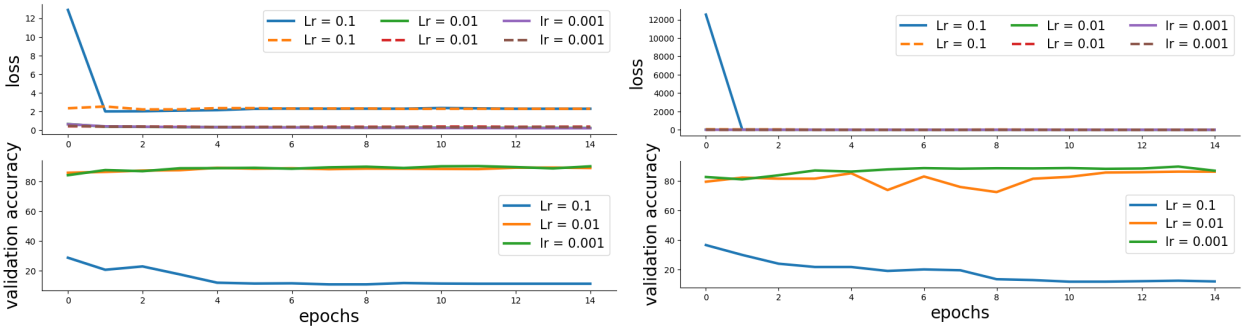
quickly, and the lower learning rates converged much more slowly and not to as high an accuracy. The loss also decreased much more quickly, as the error decreased quickly as we ran through our epochs.




(A) Baseline configuration (B) SGD optimizer under lr

FIGURE 4. SGD optimizer under baseline / varying lr conditions

Next, we tested both Adam (Figure 5a) and RMSProp (Figure 5b) optimizers. The Adam optimizer only yielded good results for low learning rates, so we can conclude that the higher learning rate caused instabilities in the model. This is supported as well by the validation accuracy decreasing as we travel through our epochs. Similarly, the RMSProp only performed well for lower learning rates as well. In our training loss diagrams, we see that the loss was so much lower for lower learning rates that we can't even see the shape of the lines, as the loss was so high for the highest learning rate.

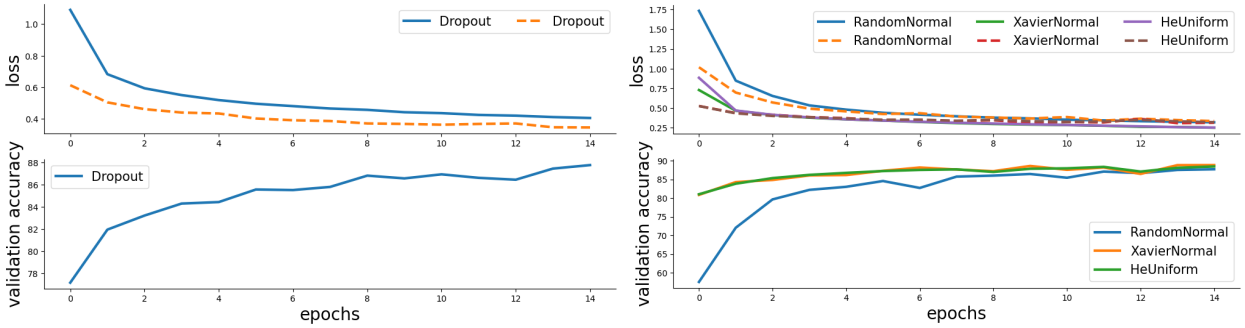



(A) Adam optimizer under lr (B) RMSProp optimizer under lr

FIGURE 6. Various optimizers under varying lr conditions

However, our testing performance was often slightly lower than the maximum training performance reached during training, implying overfitting. In our baseline test, we can see that the training loss is higher than the validation loss for many epochs, which implies overfitting. This occurred for some of our optimizers, especially Adam and SGD with low learning rates. However, Adam and RMSProp with high learning rates performed poorly on training, validation, and testing data, which could imply underfitting. However, due to the shape of the validation accuracy curves, which got even worse over more epochs, this is likely more a fault of the optimizer/lr combination.

To attempt to fix the overfitting, we used dropout regularization to attempt to improve generalizability (Figure 7a) of our model. Our particular instance used a 50% dropout rate, so at each iteration through our layers we will lose 50% of our data. This did yield a high accuracy, which is impressive considering we had such a high dropout rate.




(A) Dropout regularization (B) Different initializers

FIGURE 8. Regularization and initialization configurations

We also tested different initial configurations for our weights distributions. A Random Normal initialization took longer to converge to a higher validation accuracy, but the Xavier Normal and Kaiming (He) Uniform initializations both work consistently well, all around the same level as our baseline. Thus the He Uniform and the Xavier normal seem to be better for speed of convergence, but all initializers ended with a similar testing accuracy.

Finally, we attempted batch normalization (Figure 9), such that the output at each layer is normalized to reduce any strong features taking over the predictive power. This didn't end up improving the model fit overall, though it yielded a more interesting validation accuracy graph. This makes sense as the model would be better or worse depending on how severely the normalization affects our data.

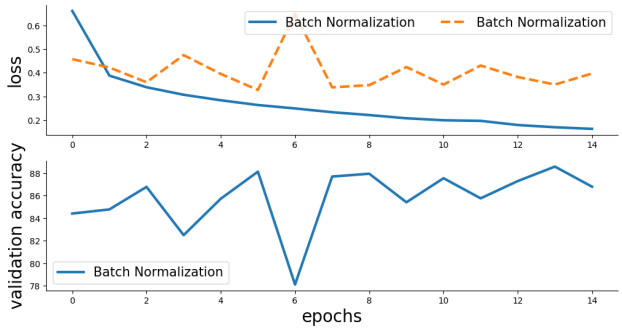
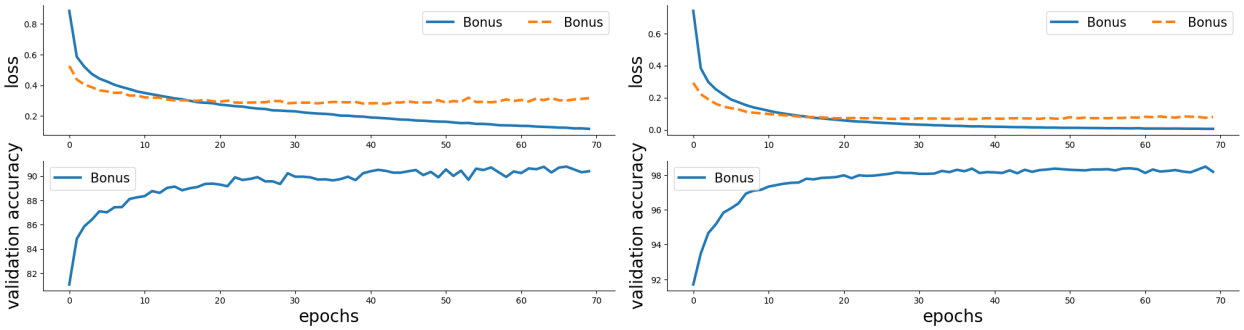



FIGURE 9. Batch normalization

The particular testing accuracy and standard deviation for each configurations are displayed in Table 1 below.

Finally, we performed additional hyperparameter tuning to achieve a testing accuracy of .9011, which occurred under the follow configurations: three hidden layers, with node sizes 700, 200, and 50; lr = .0001, RMSProp optimizer, dropout of 15%, He Uniform initializer, and no normalization. We also applied this configuration to the MNIST dataset, which yielded a testing accuracy of 0.9818 with a standard deviation of 0.00015.

We also applied the most successful classifier from assignment 3 on our FashionMNIST dataset. This was the SVD classifier, which was used for the extra credit on assignment 3, which yielded a testing accuracy of 0.9128.




(A) Fashion MNIST (B) Bonus conditions

FIGURE 10. Bonus conditions for Fashion and MNIST

| Base? | Lr | Optimizer | Dropout | Initializer | Normalization | Accuracy | SD |
|---|---|---|---|---|---|---|---|
| Y | 0.1 | SGD | None | Baseline | None | 0.8764 | 0.0005 |
| Y | 0.1 | SGD | None | Baseline | None | 0.8507 | 0.0005 |
| Y | 0.1 | Adam | None | Baseline | None | 0.1135 | 0.0004 |
| Y | 0.1 | RMSProp | None | Baseline | None | 0.1162 | 0.0010 |
| Y | 0.01 | SGD | None | Baseline | None | 0.8240 | 0.0008 |
| Y | 0.01 | Adam | None | Baseline | None | 0.8774 | 0.0006 |
| Y | 0.01 | RMSProp | None | Baseline | None | 0.8505 | 0.0006 |
| Y | 0.001 | SGD | None | Baseline | None | 0.6722 | 0.0011 |
| Y | 0.001 | Adam | None | Baseline | None | 0.8878 | 0.0004 |
| Y | 0.001 | RMSProp | None | Baseline | None | 0.8577 | 0.0010 |
| Y | 0.1 | SGD | 0.5 | Baseline | None | 0.8666 | 0.0008 |
| Y | 0.1 | SGD | None | Random Normal | None | 0.8671 | 0.0008 |
| Y | 0.1 | SGD | None | Xavier Normal | None | 0.8756 | 0.0008 |
| Y | 0.1 | SGD | None | He Uniform | None | 0.8716 | 0.0008 |
| Y | 0.1 | SGD | None | Baseline | Batch | 0.8600 | 0.0010 |
| N | 0.0001 | RMSProp | 0.15 | He Uniform | None | 0.9011 | 0.0006 |

TABLE 1. Validation accuracy scores (with standard deviation) in different contexts

## 5. Summary and Conclusions

Over the course of this project, we evaluated different combinations of hyperparameters and configurations to see which particular combination would work the best. At the end of our analysis, we found that the best model configuration to classify the FashionMNIST dataset is the Adam optimizer with a learning rate of 0.001. Further analysis could involve applying various configurations mixing regularization, initialization, and choice of optimizer for even higher classification accuracy.

## Acknowledgements

The author is thankful to our peers Laura Pong, for help creating a customizable model function for various hidden layer amounts, as well as Kaillah Selvaretnam, Heidi Neuman, Kate Everling, and Quinn Kelly for conceptual help surrounding the problem and cross-referencing algorithms and results.

## References

[1] C. R. Harris, K. J. Millman, S. J. der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, SebastianBerg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020.

[2] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[3] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[4] M. Waskom, O. Botvinnik, D. O'Kane, P. Hobson, S. Lukauskas, D. C. Gemperline, T. Augspurger, Y. Halchenko, J. B. Cole, J. Warmenhoven, J. de Ruiter, C. Pye, S. Hoyer, J. Vanderplas, S. Villalba, G. Kunter, E. Quintero, P. Bachant, M. Martin, K. Meyer, A. Miles, Y. Ram, T. Yarkoni, M. L. Williams, C. Evans, C. Fitzgerald, Brian, C. Fonnesbeck, A. Lee, and A. Qalieh. mwaskom/seaborn: v0.8.1 (september 2017), Sept. 2017.